

Fig. 1

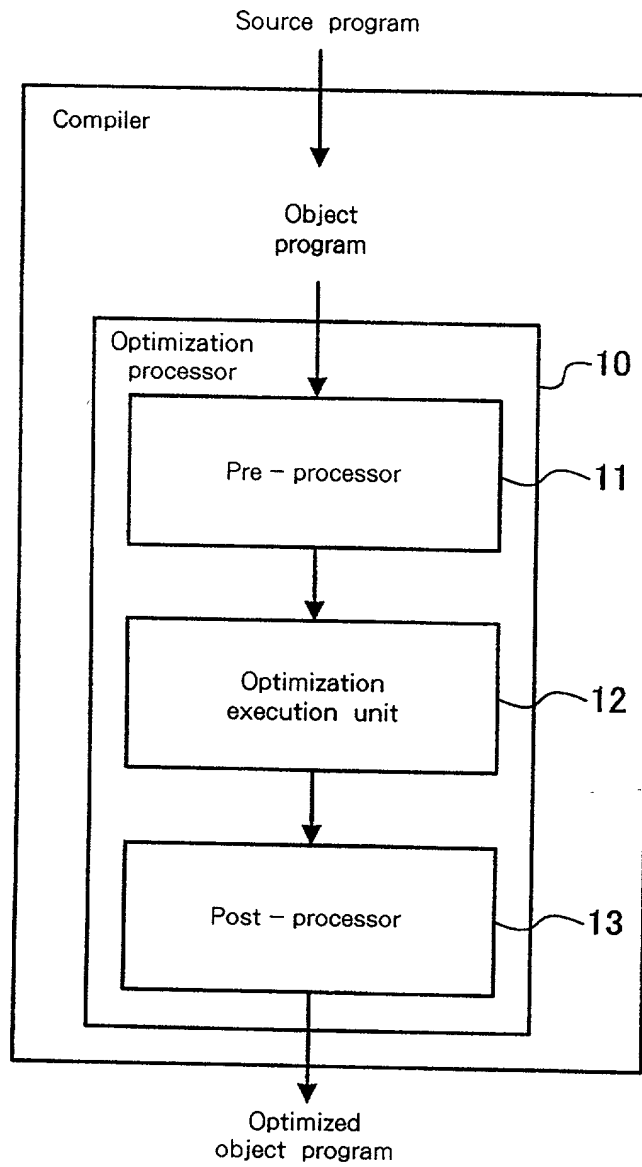


Fig. 2

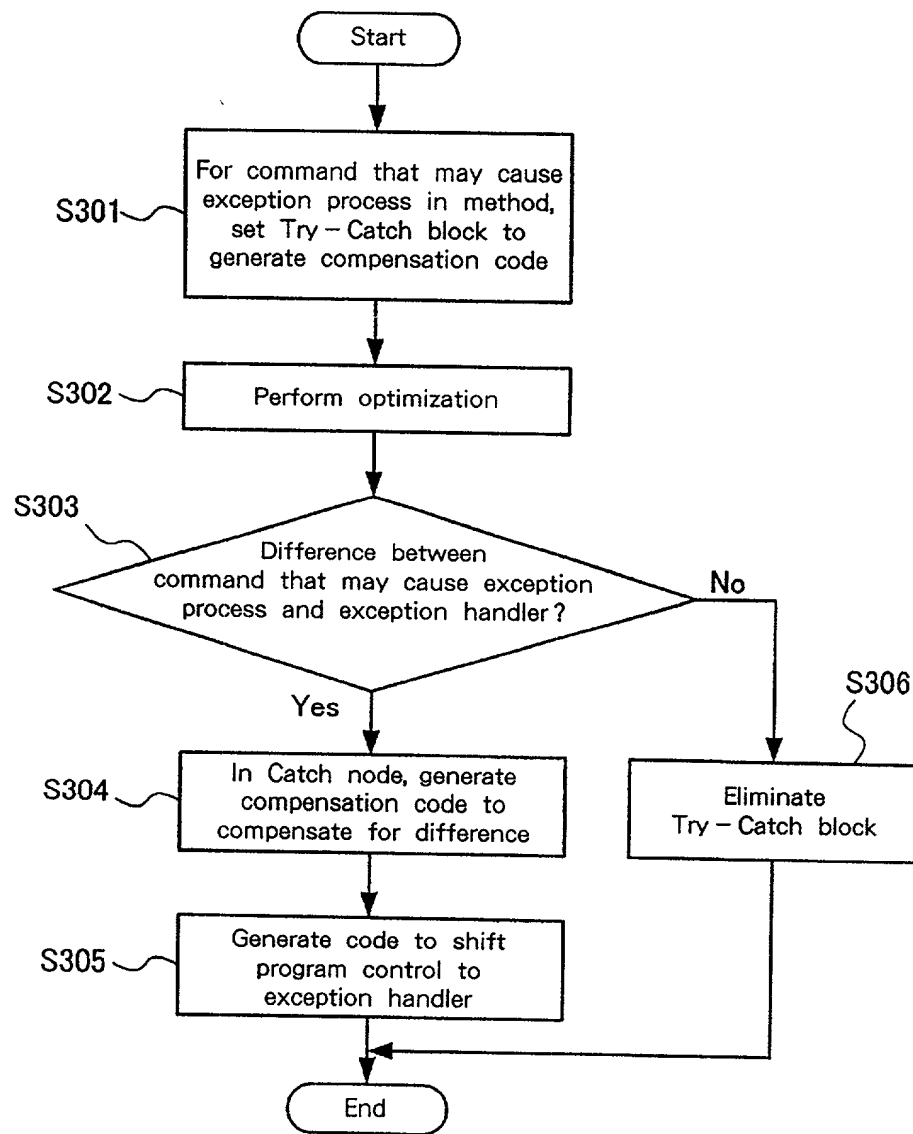


Fig. 3

```
for (repeat all basic blocks) {  
  bidx = index of basic block  
  if (this basic block is the originating point for an exception handler) {  
    reach[bidx].gen = (all TRYs and all local variables corresponding  
                      to the exception handler);  
  } else {  
    reach[bidx].gen =  $\phi$   
  }  
  reach[bidx].kill =  $\phi$   
  for (repeat all commands in the basic block in the execution order) {  
    if (command == write to local variable) {  
      reach[bidx].kill  $\cup$  = (all TRYs and written local variable numbers);  
    }  
  }  
}
```

Fig. 4

```
reach[B].in = (  $\bigcup_{P \in \text{Pred}(B)} \text{reach}[P].\text{out}$  )  $\cup$  reach[B].gen  
reach[B].out = reach[B].in - reach[B].kill
```

Fig. 5

```

for (repeat all TRYs) USE_VIA_EH_LIST[TRY] =  $\phi$ ;
for (repeat all basic blocks) {
  bbidx = index of basic block;
  if (reach[bbidx].in !=  $\phi$ ) {
    for (repeat all commands in the basic block in the execution order) {
      switch (command) {
        case write to a local variable :
          reach[bbidx].in -= (all TRYs and written local variable numbers);
          break;
        case read from a local variable :
          for (repeat all TRYs) {
            if ((TRY, a read local variable number)  $\in$  reach[bbidx].in) {
              USE_VIA_EH_LIST[TRY] U= read local variable number;
            }
          }
          break;
      }
    }
  }
}

```

Fig. 6

```

for (repeat all basic blocks) {
  if (basic block is included in a try region) {
    TRY = identification number for a Try Region
    if (USE_VIA_EH_LIST[TRY] !=  $\phi$ ) {
      bbidx = index of basic block;
      for (repeat all commands in the basic block in the execution order) {
        if (command  $\in$  command that may cause exception) {
          if (exception handler is present that corresponds to exception that may occur) {
            < set new exception handler for this command >
          }
        }
      }
    }
  }
}

```

Fig. 7

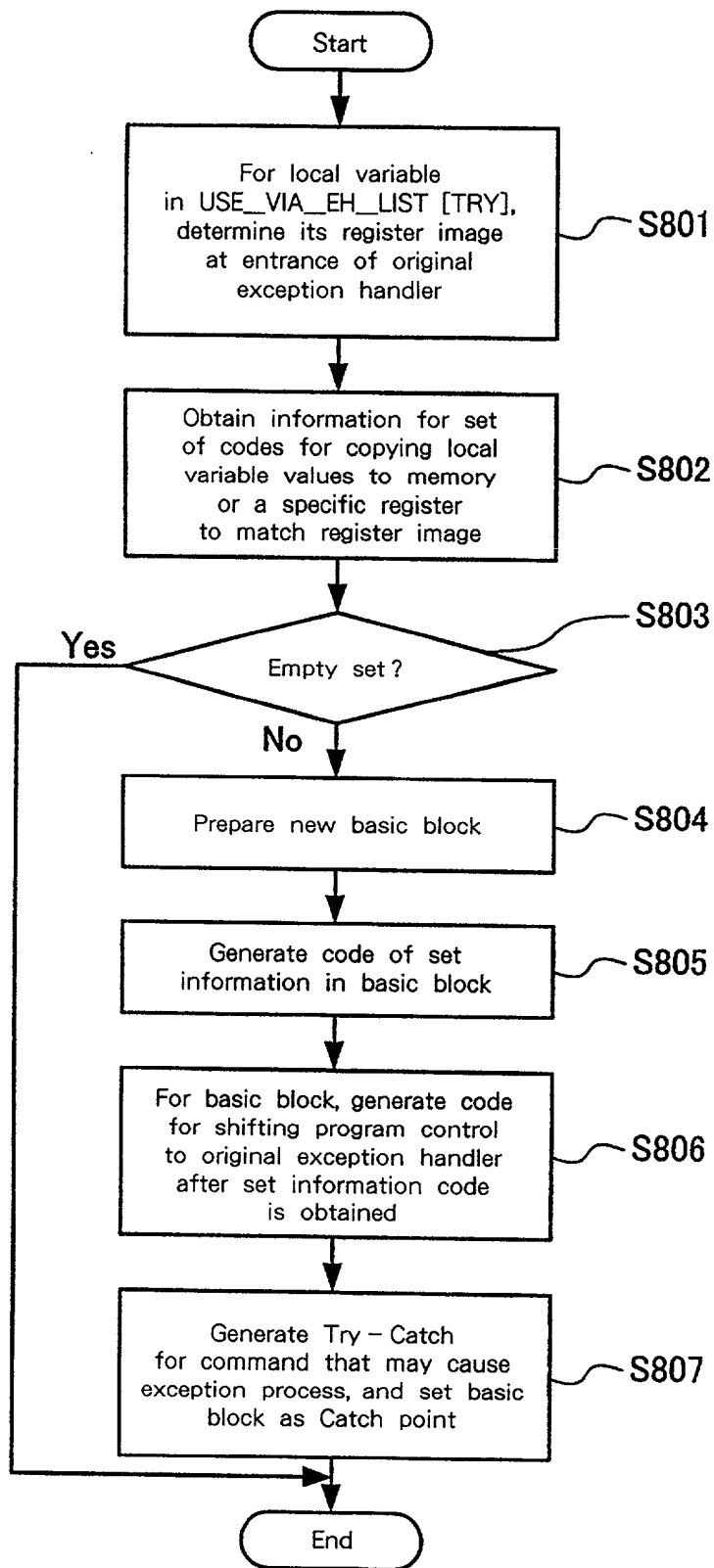


Fig. 8

```
int MIN_VAL, MAX_VAL;
void sample(int a[], int size_x, int size_y) {
    int min, max;
    int i, j;
    i = 0;          --- (1)
    j = 0;          --- (2)
    try {
        min = a[i][j]; --- (3)
        max = min;    --- (4)
        i = 0;        --- (5)
        while(i < size_x) { --- (6)
            j = 0;    --- (7)
            while(j < size_y) { --- (8)
                int val = a[i][j]; --- (9)
                if (min > val) { --- (10)
                    min = val; --- (11)
                }
                if (max < val) { --- (12)
                    max = val; --- (13)
                }
                j++; --- (14)
            }
            i++; --- (15)
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("ArrayIndexOutOfBoundsException i=" + i + " j=" + j); --- (16)
        /* error status : min > max */
        max = 0x80000000; --- (17)
        min = 0x7FFFFFFF; --- (18)
    }
    MIN_VAL = min; --- (19)
    MAX_VAL = max; --- (20)
}
```

Fig. 9

```

int MIN_VAL, MAX_VAL;
void sample(int a[], int size_x, int size_y) {
    int min, max;
    int i, j;
    i = 0;          — (1)
    j = 0;          — (2)
    try {
        NULLCHECK a;          — (3.1)
        try {
            SIZECHECK i, a[]; — (3.2)
        } catch (ArrayIndexOutOfBoundsException e) {
            copy i and j variable values to R1 and R2
            goto Handler;
        }
        try {
            SIZECHECK j, a[i]; — (3.3)
        } catch (ArrayIndexOutOfBoundsException e) {
            copy i and j variable values to R1 and R2
            goto Handler;
        }
        min = a[i][j]; — (3.4)
        max = min; — (4)
        i = 0; — (5)
        while(i < size_x) { — (6)
            j = 0; — (7)
            while(j < size_y) { — (8)
                try {
                    SIZECHECK i, a[]; — (8.1)
                } catch (ArrayIndexOutOfBoundsException e) {
                    copy i and j variable values to R1 and R2
                    goto Handler;
                }
                try {
                    SIZECHECK j, a[i]; — (8.2)
                } catch (ArrayIndexOutOfBoundsException e) {
                    copy i and j variable values to R1 and R2
                    goto Handler;
                }
                int val = a[i][j]; — (9.3)
                if (min > val) { — (10)
                    min = val; — (11)
                }
                if (max < val) { — (12)
                    max = val; — (13)
                }
                j++; — (14)
            }
            i++; — (15)
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        Handler:
        /* i, j variable values are stored in R1 and R2 */
        System.err.println("ArrayIndexOutOfBoundsException i=" + i + " j=" + j); — (16)
        /* error status : min > max */
        max = 0x80000000; — (17)
        min = 0x7FFFFFFF; — (18)
    }
    MIN_VAL = min; — (19)
    MAX_VAL = max; — (20)
}

```

Fig. 10



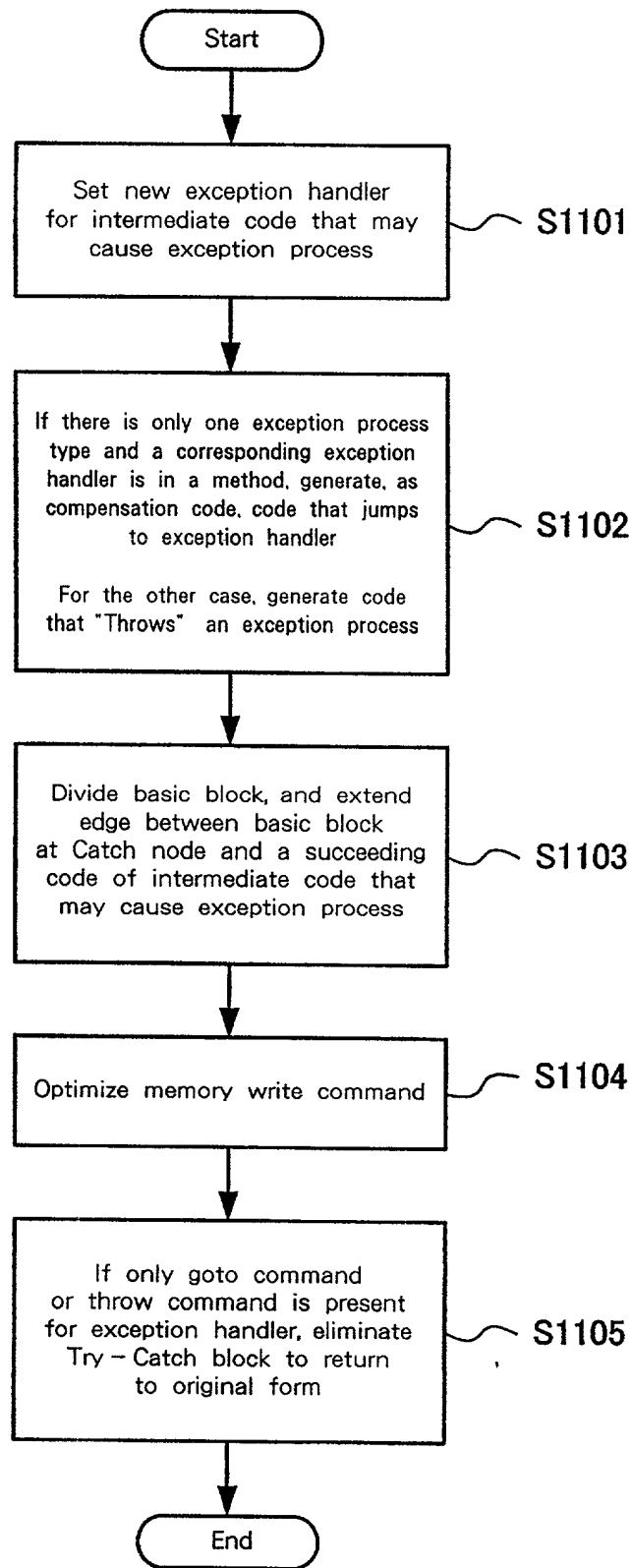


Fig. 11

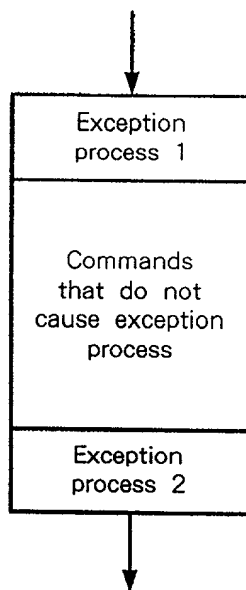


Fig. 12

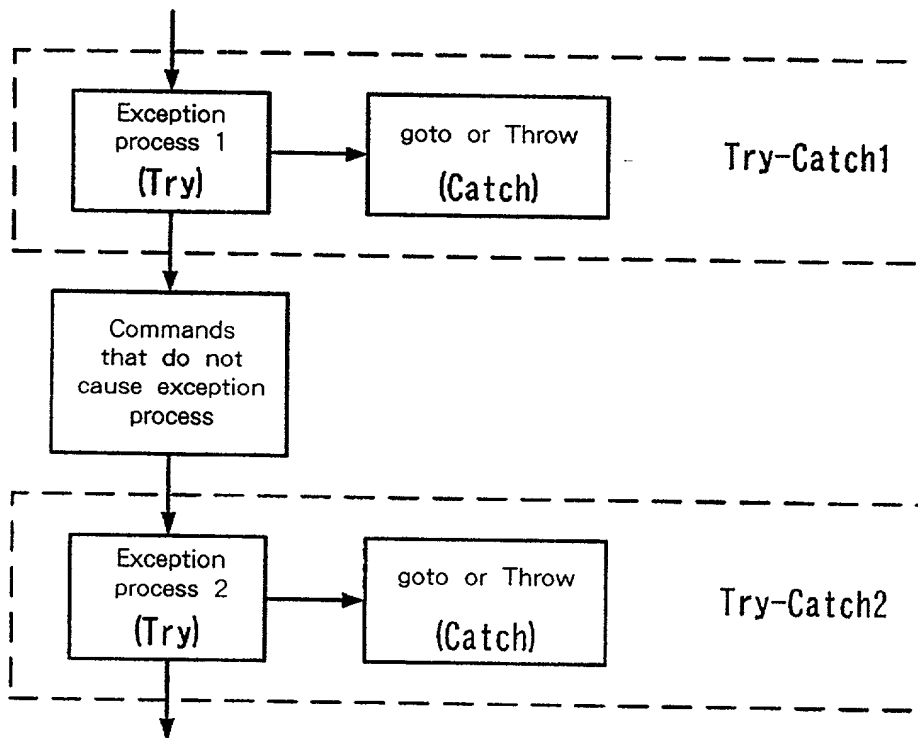


Fig. 13

```
static int mem_pos;    // variable in a memory
static int mem_a[];    // variable in a memory
void SearchPos(int reg_data)
{
    mem_pos = 0;
    while (mem_a[mem_pos] != reg_data) {
        mem_pos ++;
    }
}
```

Fig. 14

```
static int mem_pos;    // variable in a memory
static int mem_a[];    // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;        // variable in a register
    int reg_a[];        // variable in a register

    mem_pos = 0;
    goto entry;
loop:
    reg_pos = mem_pos;
    reg_pos ++;
    mem_pos = reg_pos;
entry:
    reg_pos = mem_pos;
    reg_a = mem_a;
    NULLCHECK reg_a;    // intermediate code that may cause an exception process
    SIZECHECK reg_pos, reg_a[]; // intermediate code that may cause an exception process
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

Fig. 15

```
static int mem_pos;    // variable in a memory
static int mem_a[];    // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;        // variable in a register
    int reg_a[];        // variable in a register

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    NULLCHECK reg_a;    // intermediate code that may cause an exception process

    goto entry;
loop:
    reg_pos++;
    mem_pos = reg_pos; ① // this intermediate code can not be moved outside a loop
entry:
    SIZECHECK reg_pos, reg_a[]; // intermediate code that may cause an exception process
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

1601

1602

Fig. 16

```
static int mem_pos;    // variable in a memory
static int mem_a[];    // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;        // variable in a register
    int reg_a[];        // variable in a register

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    try {
        NULLCHECK reg_a; // intermediate code that may cause an exception process
    } catch (Throwable t) {
        throw t;
    }
    goto entry;
loop:
    reg_pos++;
    mem_pos = reg_pos;
entry:
    try {
        SIZECHECK reg_pos, reg_a[]; // intermediate code that may cause an exception process
    } catch (Throwable t) {
        throw t;
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

1701

1702

Fig. 17

```

static int mem_pos;    // variable in a memory
static int mem_a[];    // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;        // variable in a register
    int reg_a[];        // variable in a register

    reg_pos = 0;
    reg_a = mem_a;
    try {
        NULLCHECK reg_a; // intermediate code that may cause an exception process
    } catch(Throwable t) {
        mem_pos = reg_pos;
        throw t;
    }

    goto entry;          1801
loop:
    reg_pos++;
entry:
    try {
        SIZECHECK reg_pos, reg_a[];    // intermediate code that may cause an exception process
        //
    } catch(Throwable t) {
        mem_pos = reg_pos;    -①
        throw t;
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
    mem_pos = reg_pos;    -②          1803
}

```

Fig. 18

```

try {
    ExceptionCheck;    // intermediate code that may cause an exception process
} catch(Throwable t) {
    throw t;
}

```

↓

```

ExceptionCheck; // intermediate code that may cause an exception process

```

Fig. 19

```
static int mem_pos;    // variable in a memory
void SearchPos(int reg_data)
{
    mem_pos = 0;
    while (func(mem_pos) != reg_data) {
        mem_pos ++;
    }
}
```

Fig. 20

```
static int mem_pos;    // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos; // variable in a register
    int reg_ret; // variable in a register

    reg_pos = 0;

    goto entry;
loop:
    reg_pos ++; 2101
entry:
    try {
        reg_ret = func(reg_pos); 2102
    } catch (Throwable t) {
        mem_pos = reg_pos;
        throw t;
    }
    if (reg_ret != reg_data) goto loop; 2103
    mem_pos = reg_pos;
}
```

Fig. 21

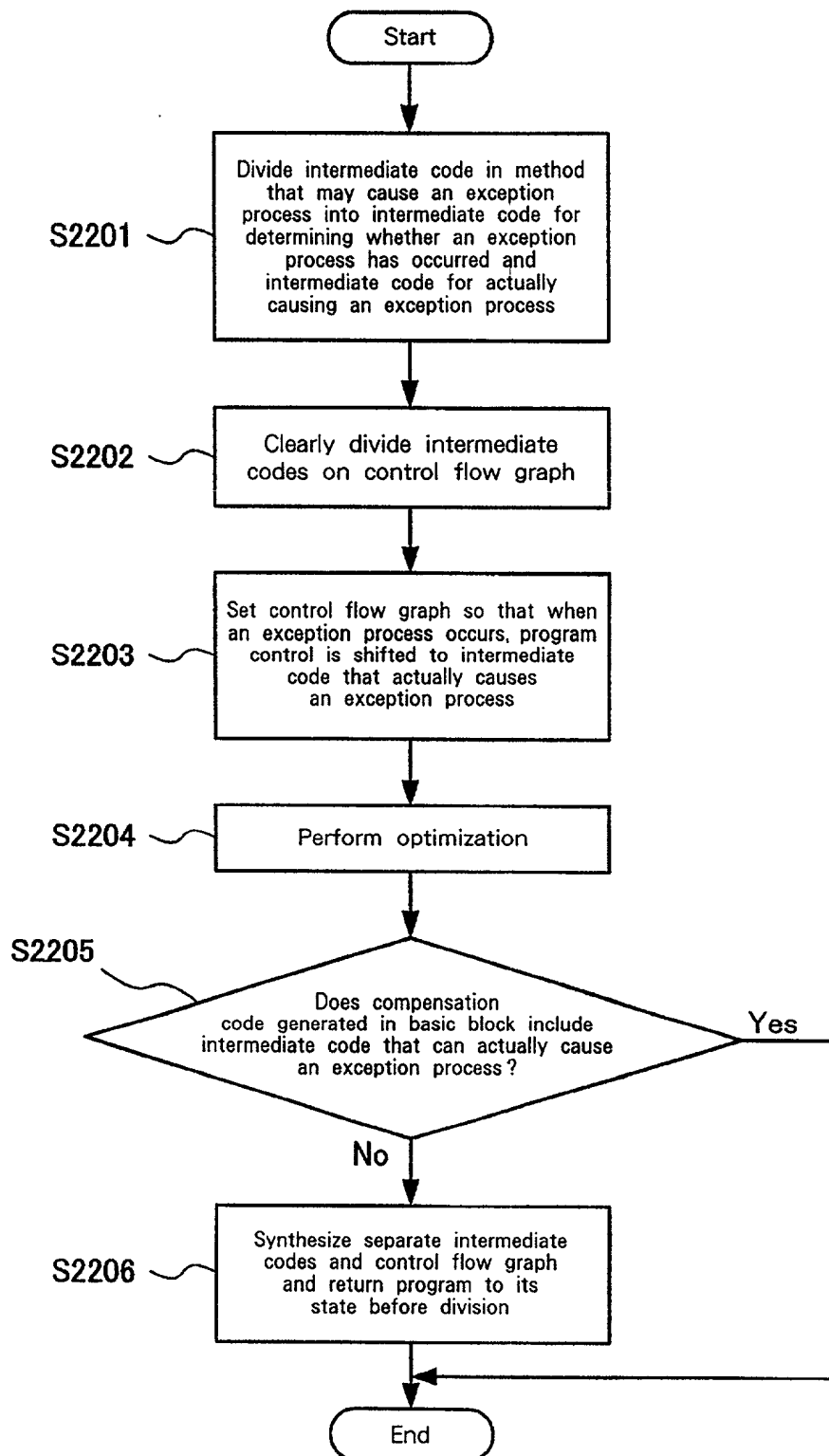


Fig. 22



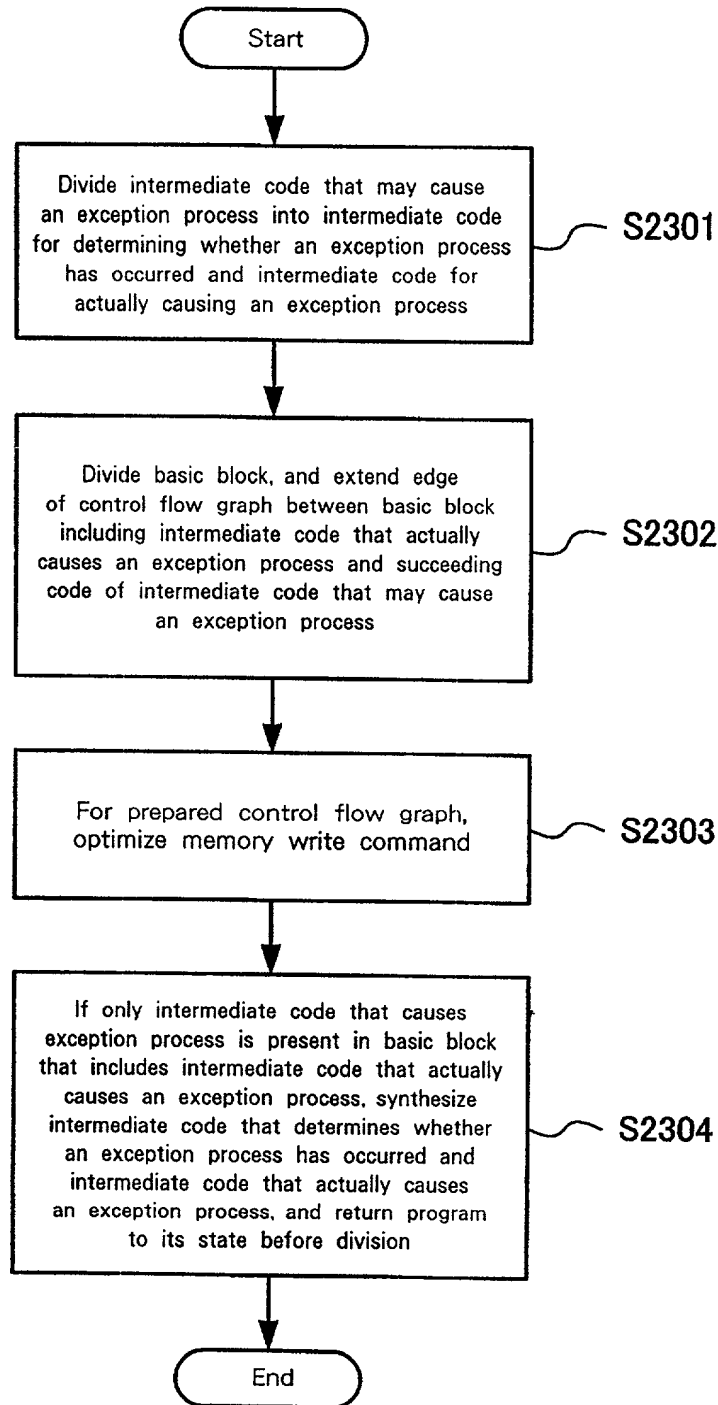


Fig. 23

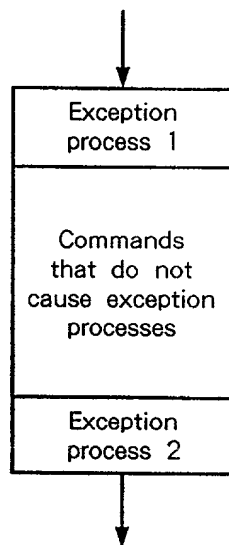


Fig. 24

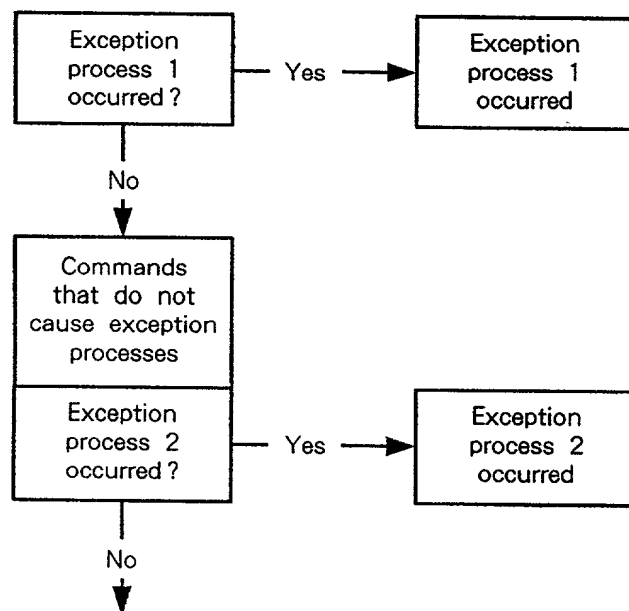


Fig. 25

```
static int mem_pos;    // variable in a memory
static int mem_a[];    // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;        // variable in a register
    int reg_a[];        // variable in a register

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    if (NULLCHECK_FAIL reg_a) { // intermediate code that determines
                                // whether an exception process will occur
        NULLCHECK_EXCEPTION reg_a; // intermediate code that causes an exception process
    }

    goto entry;
loop:
    reg_pos++;
    mem_pos = reg_pos,
entry:
    if (SIZECHECK_FAIL reg_pos, reg_a[]) { // intermediate code that determines
                                            // whether an exception process will occur
        SIZECHECK_EXCEPTION reg_pos, reg_a[]; // intermediate code that causes an exception process
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

2601

2602

Fig. 26

```

static int mem_pos;    // variable in a memory
static int mem_a[];    // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;        // variable in a register
    int reg_a[];        // variable in a register

    reg_pos = 0;
    reg_a = mem_a;
    if (NULLCHECK_FAIL reg_a) { // intermediate code that determines
                                // whether an exception process will occur
        mem_pos = reg_pos;
        NULLCHECK_EXCEPTION reg_a; // intermediate code that causes an exception process
    }

    goto entry;
loop:
    reg_pos ++; 2701
entry:
    if (SIZECHECK_FAIL reg_pos, reg_a[]) { // intermediate code that determines
                                            // whether an exception process will occur
        mem_pos = reg_pos; ①
        SIZECHECK_EXCEPTION reg_pos, reg_a[]; // intermediate code that causes an exception process
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
    mem_pos = reg_pos; ② 2703
}

```

Fig. 27

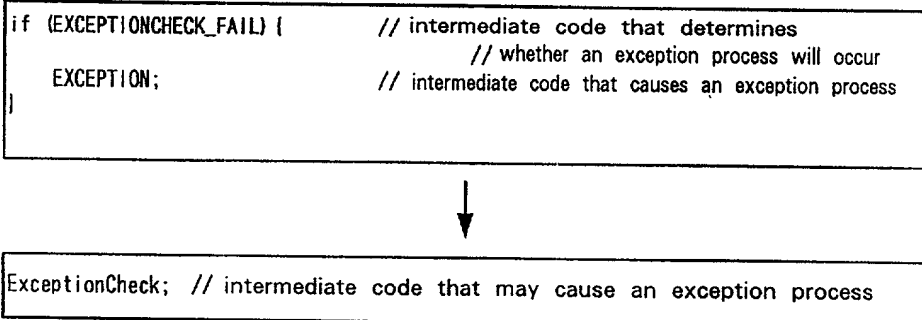


Fig. 28